新人技術者のための

ロジカル・シンキング入門 第6回

232回で止まる割り込み

冴木 元

システムの記事　ビギナーズ



ハードウェアをブラックボックス（正しく動作して当然のもの）として設計を進めるソフトウェア技術者も少なくない．ここでは，システムとして最適化されたプログラミング，あるいはデバッグの際の切り分けを容易にするため，ソフトウェア設計者にとって必要なクロックや回路の初期化/リセット，電源供給など，ハードウェア（回路）の基礎知識をまとめる．（編集部）

Hさんのチームでは，この2週間ほど陰うつな日が続いています．なぜなら，1ヵ月ほどかけて行うはずだった結合の作業が，ここのところストップしているからです．

これからの工程では，今まで単体で実装やテストを行ってきたアルゴリズムを結合し，タスク構成を順次組み上げていく予定でした．しかし，あるアルゴリズムを結合したタスクを組み上げたとたん，システム全体が動かなくなることに気づきました．最初にこの現象に気がついたとき，割り込みが232回ほど入ったときに止まることがわかりました．次に試行すると185回の，さらにその次は286回の割り込みでシステムが止まりました．どうやら，止まるまでの回数にはあまり意味がなく，「途中で止まる」とだけ表現したほうがかえって正確なようでした．

工程の期限は刻一刻と迫っています．しかし，上記のタスクが一つあるだけで全体が動かなくなるのですから，タスク構成を組み上げるどころではありません．Hさんは「テスト項目消化数ゼロ」と書かれた進捗管理表を眺めながら，ため息を漏らしました．「先週もゼロと書いて進捗会議に出た記憶があるな．でも，テスト項目の中に例のアルゴリズムが結合されないで確認できる項目などないからな…」．

いろいろとデバッグを繰り返してHさんがたどり着いた結論は，「デバイス・ドライバとして提供されているライブラリ関数の仕様書をいくら眺めても答えが出ないらしい」ということでした．ライブラリ関数ではなく，その先で制御されているハードウェア回路の仕様書を確認していかないとどこに原因があるかがわからない，ということに気づいたのです．そうして，ようやく取り寄せたハードウェア回路の仕様書を見てHさんは目がチカチカしてきました．仕様書はすべて英語で書かれていて，全体が数百ページもあったからです．Hさんは2ヵ月前の新人配属を思い出して後悔しました．「しまった，帰国子女だとかいうIをこっちに配属させておけばよかった．あいつなら訳してくれたような気がする…」．

読者のみなさんはこういうケースでどのような切りわけ方法を考えられるでしょうか．もちろん，帰国子女だというだけの理由で数百ページの仕様書を翻訳させられたらたまったものではないので，「Iさんを呼んでくる」というのは解決策に含まれないのですが…．

● 切り分けの全体像をつかむ

本誌2006年6月号（pp.50-59）で紹介した考えかたをもとに，上記のケースの原因ブロックを切り分けていくとすると，どこに着目するとよいでしょうか（**図1**）．

例えば，「アルゴリズムのせいか否か」で切り分ける，という考えかたが適応できそうです．つまり，アルゴリズムをダミーの転送処理に置き換えた場合でもシステムが止まるかとまらないかで，アルゴリズムそのものに原因がある

KeyWord　タスク構成，デバイス・ドライバ，ライブラリ関数，クロック，リセット，リリース，イネーブル，電源，グラウンド

のか，それを動かす周りの環境に原因があるのかを切り分けるというものです．ここでは，かりに**図1**の下半分のケース（アルゴリズムを動かす環境）に原因があるらしいことがわかったとしましょう．

続いて，ハードウェア回路の設定，ハンドラの実装など，一つ一つに問題がないかを確認していくことになります．

● ライブラリだけではすまないことも

ところが，そうしたハードウェアの設定を一つ一つ確認していく切り分け作業は，ソフトウェア技術者にとっては意外とやっかいな場合があります．というのは，ライブラリを使って組み上げた形で動かす方法は知っていても，一つ一つの回路がどのように設定されて動いているかを意識して使っていない人もいるからです．

近年，組み込み環境においてもライブラリ関数が提供され，アセンブラでデバイス・ドライバを組み上げなくてもよいケースが増えてきました．このことは開発効率を上げてくれるようにもみえますが，その反面，ハードウェアの設定がブラックボックス化されてしまい，切り分け作業が行いにくくなることもしばしば起こるのです．

そのようなソフトウェア技術者のために，今回はハードウェアの設定を対象とするデバッグの全体的な流れを解説することにします．

● ハードウェアはメモリ・マップド・レジスタで設定

CPUからハードウェア回路を設定する場合，ほとんどのケースでは「メモリ・マップド・レジスタ」と呼ばれるものがあらかじめ用意されていて，プログラマはそれを使うことになっています（**図2**）．これは，回路の設定を行うときに使うレジスタを，メモリのある番地に割り当てておくものです．こうすることによってプログラマは番地を指定すれば，メモリにアクセスするときと同じように周辺回路のレジスタにアクセスできるようになります．

したがって，ソフトウェアのプログラマは，ハードウェア回路を扱う際にはまず，メモリ・マップド・レジスタを調べます．今回の例では，ハードウェアの仕様書を入手したら，メモリ・マップド・レジスタにどのようなものがあって，どの番地に割り当てられているのかを最初に調べなければなりません．

● ハードウェアを単純化して見通しを良くする

メモリ・マップド・レジスタがどのようなもので，何番地に割り当てられているのかがわかれば，次にそれらをどのように設定するのかを調べます．当然，回路ごとに設定は異なりますが，ここでは「ハードウェア回路」という抽象化された観点で話を進めます．

まず信号線についてですが，ごく単純にいうとディジタル回路には**図3**に示すような，クロック，リセット，制御信号，（入出力）データ信号が存在します．それらを**図4**のように設定すると，回路を動かすことができます．ハードウェア技術者には，これは極端に単純化したものにも見えるかもしれません．しかし，ソフトウェア技術者の中には回路設計の「常識」をほとんど知らない人が数多くいます．そのため，まずは全体像を大まかにとらえる方法がないと，（筆者も含めて）どうやって切り分ければよいのかがわからない場合もあるのです．

また，ソフトウェア技術者の観点から「回路は作るものではなく使うもの」と割り切って単純化してしまうことで，システム全体の見通しがよくなることもあります．

ハードウェアを動作させるには**図4**に示したように，リ

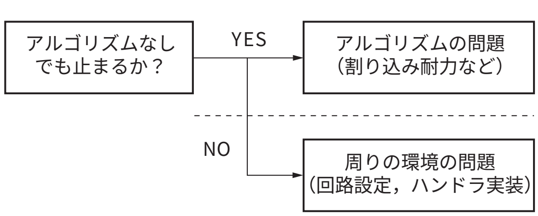


図1　切り分けの全体像

本誌2006年6月号にならって，切り分けの全体像を考えてみる．タスクの中のアルゴリズムを削除した場合，動くか動かないかで大きく分けることができる．今回のケースではおもに下にフローが流れたときに着目．一つ一つ設定を確認するのだが，ソフトウェア技術者にはわかりにくいことも….

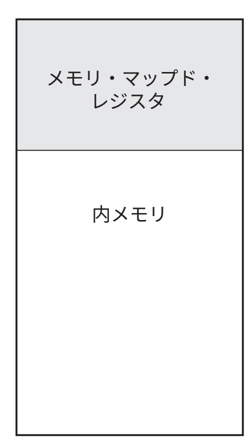


図2
回路の設定はメモリ・マップド・レジスタで

ハードウェア回路をCPUから設定する場合，メモリ・マップド・レジスタを使って設定することが多い．メモリ・マップド・レジスタは，メモリの特定番地に割り当てられたレジスタのことで，プログラマは番地を指定することでメモリと同じようにアクセスできる．

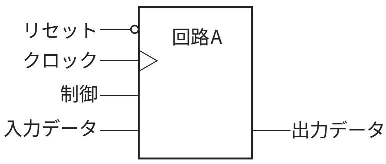


図3　回路の基本的な信号線の種類と役割
ハードウェア回路につながる信号線は大きく分けて，クロック，リセット，制御信号，（入出力）データ信号がある．実際はもっと複雑だが，大まかにあたりをつけて調べていくと，使いかたが見えてくる．

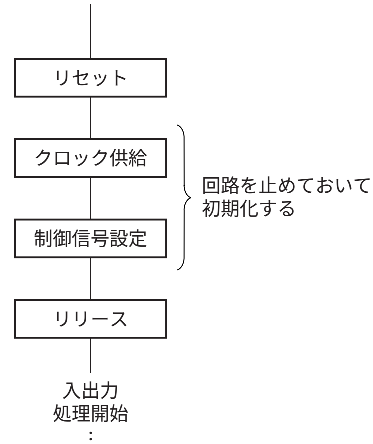


図4
回路の大まかな初期設定の流れ
大まかな流れは，回路を止めて初期化した後に処理を開始するというもの．図は，リセットとリリースをそれぞれ手動で行う場合のフローを表している．

セット，クロック供給，初期設定，処理開始という流れをたどります（ソフトウェアとの違いや注意点は後述）．この場合の初期設定とは，制御用の信号線に回路の動作パラメータを設定することです．

ここで，クロックを省いてリセットと初期化処理をまとめれば，リセット→処理開始となります．こうするとソフトウェアと同じように考えられるかもしれません（ソフトウェアの場合も基本的な処理の流れを単純化すれば，リセット→処理開始）．

しかし，少し細かくみていくと，ソフトウェアとハードウェアの考えかたの違いが現れてきます．これらを，クロックとリセットに分けて解説していくことにします．

●「クロックがないと動かない」という常識

クロックの供給には次の三つの手順を踏みます（**図5**）．

1）クロック・ソースを決める

2）分周率を決める

3）クロックをイネーブルする

クロックの配分が複雑で，いくつもの分周器やマルチプレクサによる選択を介す場合でも，基本的な考えかたは変わりません．

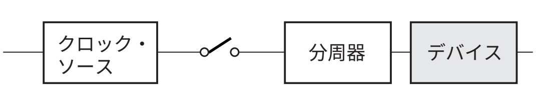


図5　回路へクロックを供給する方法
図の右端のデバイスにクロックを供給するまでがどのような設定かを考える．大まかにいうと，クロック・ソースを決めて分周率を与えればクロックは供給される．クロック線そのものがつながっていないこと（ジャンパ・ピンが外れているなど）もあるので注意．

あたりまえのことですが，ディジタル回路はクロックが供給されないと動きません[注1]．このあまりにも基本的な事がらをソフトウェア技術者は忘れている，というより知らない事すらあります（恥ずかしながら，筆者も昔はそうだった…）．

ソフトウェアによる回路の設定がいかに正しくても，クロックを供給する配線がつながっていなければ（例えば，ジャンパ・ピンが外れているなど）回路は動きません．クロックの設定が必要かどうかか，必要だとしたらどのレジスタを設定する必要があるのか，ということを最初に確認する必要があります．

● ソフト技術者が知っておくべきリセットに関する注意点

リセットは回路の中にあるフリップフロップを初期化する処理です（**図6**）．ソフトウェア的には「スタティック変数の初期化」といってもいいでしょう．ハードウェア特有の話としては以下の**1)～4)**のようなものがあります．

注1：ここでは同期型のディジタル回路を前提に話をしている．

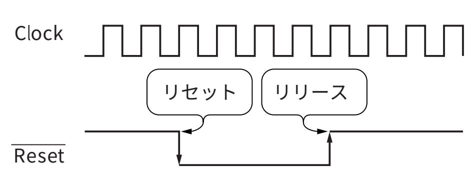


図6　「リセット」と「リリース」
「リセット」してフリップフロップを初期化した後に「リリース」する．両者を手動で行う場合，リセット中に初期設定をすませてリリースする．リセットは1クロックで終わるとは限らないため，wait（待機）処理が必要となることが多い．

1）リセットの後にリリースが必要

リセット信号は，通常の信号線と極性が逆（アクティブL；信号が'0'のときON）のケースが多いです．C言語で書くと，**図7**のいちばん上のパスがリセット処理のイメージです．

リセット処理が終わり，信号が'1'に戻ると（リリースされると）通常の処理が開始されるので，その間に必要な初期設定を行うことになります．

2）リセットは1クロック分とは限らない

リセット処理は，かならずしも1クロックで終わるとは限らないことに注意が必要です．回路内のフリップフロップをすべて初期化するのに1クロックですむとは限らないからです（大きな回路の場合，リセットですべてのフリップフロップが初期化されるわけではない）．

したがって，「リセット」してから「リリース」するまでの間にwait（待機）処理が必要なことがあります．例えば，「リセット後，リリースまで2クロック分のwaitが必要」とハードウェアの仕様書に書かれているのに気づかずにプログラムを書いてしまった場合，そのハードウェアが未定義の動作をしてしまうことがあるので注意が必要です．

3）リセット後のリリースが自動的になされることも

リリースの設定はプログラムで明示的に設定する設計もあれば，リセット後に回路が自動的に戻してくれる場合もあります．リセット後のリリースが回路で自動的に行われる場合，**2）**でお話ししたwaitは，リリースの設定レジスタの値が'0'から'1'に戻るのを待つ処理に変わります．

4）イネーブルと区別

リセットはイネーブル信号と区別する必要があります．両者には**図7**に示したような違いがあります．

リセットは内部のフリップフロップに保存されているステート情報を初期値に戻す信号のことですが，イネーブルは単に通常の処理を行うかどうかを決めるための信号です．

ディセーブル→初期化→イネーブルというシーケンスで回路の初期設定を行うことがあります．この場合のディセーブルはあくまで回路の動作を止めたにすぎないので，注意が必要です．仕様書をよく読んで，自分が設定しようとしている信号がただのイネーブルなのかリセットなのかを確認しましょう．

両者を組み合わせて使う必要がある場合などは，例えばディセーブル→リセット→初期化→イネーブルというような使いかたになると思います．

● 電源とグラウンドをソフト技術者が意識する場面

さて，**図3**には実は略している信号線があります．ハードウェア技術者ならすぐわかると思いますが，それは電源とグラウンド（GND）です（**図8**）．ソフトウェア技術者ができ上がった回路を使いこなすということに話を限定した場合，両者を意識する場面は限られてきます．

電源については，回路を省電力モード，すなわち動作周波数を落としたり，クロックを止めたりする動作を行う場合，ソフトウェア技術者といえども電源の使いかたを考えなければなりません．回路を省電力モードにするとき，通常動作に戻すときなどは，電源やクロックの供給，リセットをかける順番に気をつける必要があります．これは，それぞれの回路の仕様書を確認するしかないでしょう．

グラウンドについては，これを触った経験は筆者にはほとんどありません．ただし，製造の初期段階のボードを開発で使用しているときなどは，ときどき「グラウンドが弱

```
if(reset == 0)
{
    リセット処理
}
else
{
    if(enable == 1)
    {
       通常の処理
    }
}
```

図7　「リセット」と「イネーブル」

図の"reset"と"enable"の違いに注意．resetはほんとうに各フリップフロップを初期化するものだが，enableは単に通常の処理を止めるだけ．マニュアルをよく読んで確認すること．さらに，部分的にリセットするだけの信号線もあるので注意．

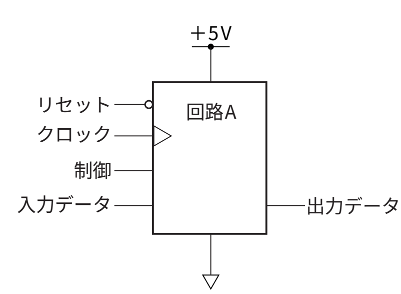


図8　電源とグラウンド

図3には実は略した信号線，すなわち電源とグラウンドが存在する．両者は回路図でも略されることが多い．ソフトウェア技術者が電源を意識するのは，主に省電力モードで回路を動かす必要がある場合などである．

すぎて動作が不安定」などというものに出くわすことがあります．このような場面の切り分けは，もはや個体不良品の切り分けになるので，ここでは省略します．

● 回路を使いこなすための具体例

以上をまとめて，冒頭の例にあったような割り込みハンドラの中にアルゴリズムを実装する場合について考えてみましょう．**図9**に示すように，「デバイス1」と呼ばれる回路の割り込みを契機としてアルゴリズムが駆動されるケースを考えてみます．

まず，デバイス1そのものの設定が必要であることはいうまでもないでしょう．このデバイスの仕様を調べてレジスタに設定を行い，動作することを確認します．

次に，デバイス1がCPUに対して外部にある場合など，割り込みをCPUにたどり着かせるためにもマルチプレクサの設定などが必要となることがあります．その場合，割り込み線をどのように割り当てればよいのかを調べます．最終的に，デバイス1の割り込みがCPUの「割り込みの何番」にたどり着くのかを調べ，実装します．

続いて，割り込みが無事CPUにたどり着くことが確認できれば，割り込みハンドラを実装します．割り込みの実装だと少し敷居が高いという場合は，まず**図10(a)**に示すポーリングで実装してみる，という段階を踏むこともできます．ポーリングというのは，回路の処理終了を知らせるレジスタのビットをプログラムのフラグで見張れるようにしておいて，フラグが切り替わるとCPUでは次の処理に進む方法です．

ここまで組み上がれば，いよいよアルゴリズムの組み込みに移ります．

図9の③までの段階のテスト・プログラムが作成済みでアルゴリズムの組み込みに移るのと，いきなりアルゴリズムの組み込みに入ってしまうのには，大きな違いがあります．例えば，**図9**の③までの（アルゴリズム未実装の）段階で1時間くらいシステムを放置しても問題なく動き続けることまで確認してから，冒頭のようなバグに出くわした場合を想像してください．この場合，バグを見つけたときに「動かなくなった原因は，どうもアルゴリズムが怪しい」と推測できると思います．なぜなら，アルゴリズムが実装されていない状態では一定時間システムが問題なく動くことがすでに確認されているのですから….

ここまで単純にいかないケースもあるでしょう．しかし，そのような場合でも回路を一つ一つ動作確認しながらプログラムを組み上げれば，途中で問題が発生しても切り分けがスムーズに進められることが少なくありません．

●「ハードウェア透過性」の落とし穴

近年は，組み込みの開発環境でもOSが整備され，周辺回路を設定する際にもC言語から使えるライブラリ関数が最初から用意されることも多くなってきました（**図11**）．そして，ゼロからプログラムを組み上げなくても，提供されたサンプル・コードを使って目的のアルゴリズムを組み込むことで，システムの統合作業を進められるようになりま

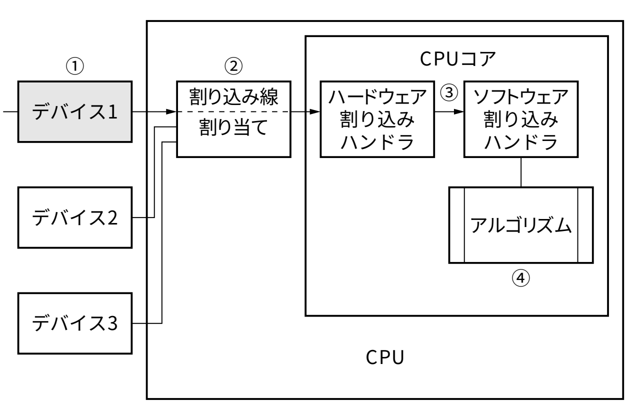


図9　割り込みハンドラの中にアルゴリズムを実装する例
図は，デバイス1の割り込みでアルゴリズムが起動する実装の例．①デバイス1そのものの設定，②外部回路から割り込み信号が届く経路の確保，③ハンドラの実装，④アルゴリズムの組み込み，がそれぞれ必要．

```
volatile int flg;
...
while(!flg){}

アルゴリズムCall
```

（a）ポーリング

```
xx_handler()
{
    アルゴリズムCall
}
```

（b）割り込み

図10　アルゴリズムを通信回路と同期させて動かす2種類の方法
（a）は処理終了をポーリングする例，（b）は割り込みハンドラを実装する例．割り込みハンドラの実装が最終的な目標でも，切り分けのために，まず（a）のように実装してみることもある．

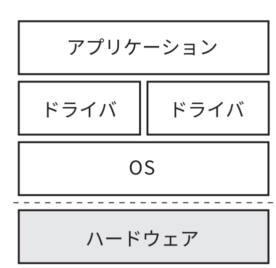


図11　透過性の落とし穴
組み込み環境でもOSが整備され，アプリケーションがハードウェアを意識しないですむ環境が増えてきた．ところが，実際にはしばしばハードウェアの設定を意識せざるをえないことがあり，ブラックボックス化が逆に見通しを悪くすることもある．

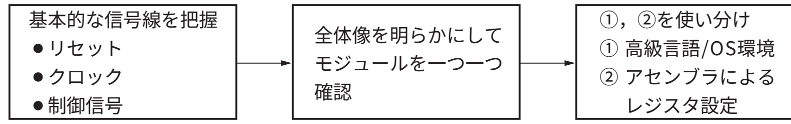


図12
ハードウェア・デバッグの全体像
ソフトウェア技術者は，大まかにでもハードウェアの信号線の基本を理解する必要がある．全体像を明らかにして一つ一つモジュールを組み上げる．ハードウェア透過性の理想と現実も見分けること．

した．ハードウェアを意識せずにソフトウェアを開発できる環境が次第に整いつつあるようにも思えます（これらをまとめて，「ハードウェア透過性の向上」と呼ぶことができる）．

これは歓迎すべきことです．ハードウェア，ソフトウェアともに複雑化した現在の組み込み開発において，ひと昔前のように周辺回路を一つ一つ動かしてシステムを組み上げていたのでは，納期に間に合わないことが多々あるからです．

ただしこれには，提供されるライブラリやそれによって動く周辺回路が，「完ぺきに設計されていた場合」という大前提が必要となります．組み込みの開発環境においては現実問題として，不安定な，つまりバグのあるソフトウェアやハードウェアが提供されてくることが少なからずあります．また，開発環境の品質がそこまでひどくなくても，わかりにくい動作条件件を持った回路に出くわすことはしばしばあります．

● 小さく作って積み上げること

たいせつなことは，でき合いのサンプル・コードなどを流用する場合も，必要に応じてなるべく「小さな単位」に戻ることができる環境や考えかたを備えておくべき，ということになると思います．**図9**に挙げたような順番で回路をつなげていかなくても，いきなりアルゴリズムの組み込みに移れるようなサンプル・コードが最初から提供されていれば，それに越したことはありません．しかし，その場合でもケース・バイ・ケースで，それぞれの回路がどのように設定され，どのように使われているかを理解しておく必要があるでしょう．

一つ一つの回路には，回路が正しく動作しているかを確認するデバッグ用のレジスタが用意されていることがあります．**図9**の例で「割り込みがCPUまでたどり着かない」などという場合，途中の回路の確認用レジスタを一つ一つチェックすることで割り込み信号がどこで止まっているのかがわかることがあります．

● アセンブラと高級言語を使い分けて

パソコン用のOSを組み込み環境に移植したものなどは，「すべてはファイル」という設計思想で作られているものが少なくありません．この設計思想は，ファイル・ハンドラを取得して初期化関数を呼ぶという一連の手続きを踏めば，制御する対象がハード・ディスク装置であろうが，ソケットであろうが，プログラマはあまり意識することなく制御処理が書けるという利点があります．

しかし，この設計思想がかえって見通しを悪くすることもあります．例えば，デバッグのためにOSのライブラリそのもののコードを追ってみたりすると，コール・シーケンスの階層構造があまりに深すぎて見通しが悪く閉口することがあります．そのようなときに，ハードウェアの制御をアセンブラで直接書いてみたら，ひと目で見通せる程度の規模になってしまってあぜんすることも少なくありません．

ハードウェアを完全にブラックボックス化できない，というのは組み込みの開発環境ではまだしばらく消えない傾向のような気がします（**図12**）．

さえき・はじめ

<筆者プロフィール>
冴木元．システム・エンジニア．最近，毎週ジョギングしているのだが，たいていルーム・ランナでこなしてしまうのが，難点か…．距離が長くなると車輪をまわすネズミのような気分になってくる．秋になったら久しぶりに遠出してみるか，と思うきょうこのごろ．